

ESTRUCTURAS DE DATOS



Tipos de estructuras de datos en Python

Las **variables** nos permiten almacenar un dato temporalmente para usarlo en nuestro código.

Las **estructuras de datos** nos permiten almacenar agrupaciones/colecciones de datos para acceder a los mismos de manera sencilla y dinámica.

Python incorpora una serie de estructuras de datos predefinidas por defecto:

- Listas
- Tuplas
- Dicionarios
- Sets


ESTRUCTURAS DE DATOS: LISTAS



Listas

Una **lista** es una colección de datos **mutable**, es decir, que permite añadir, modificar y eliminar elementos. Los elementos están ordenados. También permite elementos duplicados.

Es la estructura de datos que más se utiliza.

- 
1. Crear una lista
 2. Acceso a los elementos de una lista
 - 2.1 Acceder con el índice
 - 2.2 Acceder con índices negativos
 - 2.3 Acceder con rangos de índices
 3. Modificar valores
 4. Verificar si existe un elemento usando el operador `in`
 5. Obtener el número de elementos usando la función `len()`
 6. Añadir nuevos elementos `append()`
 7. Eliminar elementos
 - 7.1 Método `remove()`
 - 7.2 Método `pop()`
 - 7.3 `del`
 8. Borrar una lista
 9. Vaciar una lista
 10. Copiar una lista
 11. Unir dos listas
 - 11.1 operador `+`
 - 11.2 `append()`
 - 11.3 `extend()`
 12. El constructor `list()`
 13. Invertir el orden de los elementos de una lista con `reverse()`
 14. Contar el número de veces que aparece un elemento en una lista con `count()`
 15. Ordenar una lista con el método `sort()`
 16. Averiguar el índice de un elemento

Listas: Creación y acceso de elementos

Para **crear** una lista utilizamos:

corchetes []

Dentro estarán los elementos del tipo de datos que sea separados por comas.

Para acceder a un elemento de la lista utilizamos su posición, es decir, su **índice**.

```
# 1. Para crear una lista utilizamos corchetes []  
# y dentro introducimos los elementos  
cars = ["tesla", "rolls", "ferrari"]  
print(cars)
```

```
['tesla', 'rolls', 'ferrari']
```

```
# 2. Acceso a elementos  
# 2.1 Para acceder a un elemento de la lista lo hacemos  
# a través de su índice entre los corchetes,  
# empezando siempre en 0  
cars = ["tesla", "rolls", "ferrari"]  
print(cars[0])  
print(cars[1])  
print(cars[2])
```

```
tesla  
rolls  
ferrari
```

Listas: acceso de elementos

Otra forma de acceder a los elementos es mediante **índices negativos**.

Si tratamos de acceder a una posición fuera de la longitud de la lista se producirá error.

```
# 2.2 También podemos acceder a los elementos haciendo uso de índices negativos
# De forma que -1 accede al 1 elemento empezando por el final
# De esta forma -2 accede al 2 elemento elemento por el final y así
cars = ["tesla", "rolls", "ferrari"]
print(cars[-1])
print(cars[-2])
print(cars[-3])

# Si intentamos acceder a un índice que no existe, como por ejemplo 4
# Se produce un error IndexError: list index out of range
# print(cars[4])
```

```
ferrari
rolls
tesla
```

Listas: acceso de elementos

Otra forma de acceder a elementos de una lista es mediante **rangos de índices**, de esta forma podemos obtener sublistas de elementos desde una posición hasta otra posición determinadas.

```
# 2.3 Otra forma de acceder es mediante rangos de índices
#           0         1         2         3         4         5         6
# Negativo: -7        -6        -5        -4        -3        -2        -1
cars = ["tesla", "rolls", "ferrari", "jaguar", "maserati", "audi", "porsche"]
# Desde el primer índice indicado (incluido) hasta el último indicado (no incluido)
print(cars[2:5])
# Si no se especifica índice inicial por defecto se utilizará el primero de todos, el 0
print(cars[:4])
# Si no se especifica índice final por defecto se utilizará el último incluido
print(cars[2:])
# También aplican los índices negativos en los rangos.
# Acordarse de que el segundo índice, el final, no está incluido
print(cars[-4:-1])
```

```
['ferrari', 'jaguar', 'maserati']
['tesla', 'rolls', 'ferrari', 'jaguar']
['ferrari', 'jaguar', 'maserati', 'audi', 'porsche']
['jaguar', 'maserati', 'audi']
```

Listas: modificar una lista

Para **modificar** una lista simplemente asignamos un nuevo valor a una posición.

Podemos comprobar la existencia de un elemento en una lista con el operador de membresía **in**.

```
# 4. Comprobar si un elemento existe en una lista.  
# Se usa el operador in  
cars = ["tesla", "rolls", "ferrari"]  
check = "tesla" in cars  
print(check)  
print("tesla" in cars)
```

```
True  
True
```

```
# 3. Modificar la lista, podemos cambiar el elemento que  
# queramos especificando su índice  
cars = ["tesla", "rolls", "ferrari"]  
print(cars)  
cars[1] = "mercedes"  
print(cars)
```

```
['tesla', 'rolls', 'ferrari']  
['tesla', 'mercedes', 'ferrari']
```

```
# 5. Obtener la longitud de la lista.  
# Se usa el método len()  
cars = ["tesla", "rolls", "ferrari"]  
print(len(cars))
```

```
3
```


Listas: añadir nuevos elementos a la lista

Existen diferentes formas de añadir nuevos elementos a una lista.

Con el método **append()** el elemento se añade al final de la lista en la última posición.

Con el método **insert()** podemos especificar como primer argumento en qué posición queremos insertar el elemento.

```
# 6. Añadir elementos a la lista.  
# 6.1 Se usa el método append()  
cars = ["tesla", "rolls", "ferrari"]  
cars.append("mercedes")  
print(cars)
```

```
[ 'tesla', 'rolls', 'ferrari', 'mercedes' ]
```

```
# 6. Añadir elementos a la lista.  
# 6.2 Se usa el método insert()  
cars = ["tesla", "rolls", "ferrari"]  
cars.insert(1, "mercedes")  
print(cars)
```

```
[ 'tesla', 'mercedes', 'rolls', 'ferrari' ]
```

Listas: borrar elementos de una lista

El método **remove()** nos permite borrar elementos.

El método **pop()** permite borrar un elemento por su índice o el último si no se especifica cada.

La palabra **del** permite borrar una lista.

```
# 7. Borrar elementos de la lista
# 7.1 Método remove()
cars = ["tesla", "rolls", "ferrari"]
cars.remove("rolls")
print(cars)
```

```
['tesla', 'ferrari']
```

```
# 7. Borrar elementos de la lista
# 7.2 Método pop(): Eliminar el último elemento
# o un elemento especificando su índice
cars = ["tesla", "rolls", "ferrari"]
cars.pop()
print(cars)
cars = ["tesla", "rolls", "ferrari"]
cars.pop(1)
print(cars)
```

```
['tesla', 'rolls']
['tesla', 'ferrari']
```

```
# 7.3 del: Eliminar un elemento por su índice
cars = ["tesla", "rolls", "ferrari"]
del cars[0]
print(cars)
```

```
['rolls', 'ferrari']
```

Listas: borrar y vaciar una lista

Una vez eliminada una lista no será accesible de nuevo a menos que se vuelva a crear con el mismo nombre.

Para vaciar una lista utilizamos el método **clear()**, a diferencia de **del** la lista sigue existiendo.

```
# 8. del: eliminar la lista
cars = ["tesla", "rolls", "ferrari"]
del cars
print(cars)
```

```
-----
NameError                                Trac
<ipython-input-23-2e5c20a33a2f> in <module>()
      2 cars = ["tesla", "rolls", "ferrari"]
      3 del cars
----> 4 print(cars)

NameError: name 'cars' is not defined
```

```
# 9. Vaciar una lista
cars = ["tesla", "rolls", "ferrari"]
print(cars)
cars.clear()
print(cars)
```

```
[ 'tesla', 'rolls', 'ferrari' ]
[ ]
```

Listas: copiar listas, unir listas

Para unir listas utilizamos:

- Operador +
- Método append()
- Método extend()

```
# 10. Copiar una lista
cars = ["tesla", "rolls", "ferrari"]
listCopy = cars.copy()
print(listCopy)
```

```
['tesla', 'rolls', 'ferrari']
```

```
# 11. Unir dos listas diferentes
# 11.1 con el operador +
list1 = ["a", "b", "c"]
list2 = [1, 2, 3]
list3 = list1 + list2
print(list3)
```

```
['a', 'b', 'c', 1, 2, 3]
```

```
# 11.2 Unir dos listas diferentes con el método append()
list1 = ["a", "b", "c"]
list2 = [1, 2, 3]
list1.append(list2[0])
list1.append(list2[1])
list1.append(list2[2])
print(list1)
```

```
['a', 'b', 'c', 1, 2, 3]
```

```
# 11.3 Unir dos listas diferentes con el método extend()
list1 = ["a", "b", "c"]
list2 = [1, 2, 3]
list1.extend(list2)
print(list1)
```

```
['a', 'b', 'c', 1, 2, 3]
```

Listas: uso de constructor y el método reverse()

Otra forma de crear una lista es siguiendo los principios de la programación orientada a objetos mediante **constructores**.

```
# 12. Constructor lista
cars = list(("tesla", "rolls", "ferrari"))
print(cars)
```

```
['tesla', 'rolls', 'ferrari']
```

Para invertir el orden actual de los elementos en una lista se utiliza el método **reverse()**.

```
# 13. Método para invertir el orden de los elementos en una lista
list1 = ["a", "b", "c"]
print(list1)
list1.reverse()
print(list1)
```

```
['a', 'b', 'c']
['c', 'b', 'a']
```

Listas: count() y sort()

El método **count()** nos permite averiguar el número de veces que un elemento se repite en una lista.

El método **sort()** nos permite ordenar la lista en orden ascendente, es decir, un orden que sigue una secuencia de menor a mayor.

```
# 14. El método count() nos devuelve el número de veces
# que un valor aparece en una lista
list1 = ["a", "b" , "c"]
print(list1.count("a"))
list1.append("a")
list1.append("a")
print(list1.count("a"))
print(list1)
```

```
1
3
['a', 'b', 'c', 'a', 'a']
```

```
# 15. El método sort() ordena una lista
# en orden ascendente por defecto
list1 = ["b", "z" , "d", "a", "f", "x"]
print(list1)
list1.sort()
print(list1)

list1 = [5, 1 , 3, 1000, 599, 600]
print(list1)
list1.sort()
print(list1)
```

```
['b', 'z', 'd', 'a', 'f', 'x']
['a', 'b', 'd', 'f', 'x', 'z']
[5, 1, 3, 1000, 599, 600]
[1, 3, 5, 599, 600, 1000]
```

Listas: el método index()

El método **index()** nos permite averiguar el índice o posición de un elemento en una lista.

Si el elemento que pasamos a **index()** como argumento no existe entonces se producirá una **excepción**.

```
# 16. También podemos averiguar el índice de un elemento a partir de su valor
# IMPORTANTE: solo devuelve el índice de la primera ocurrencia encontrada
list1 = ["b", "z", "d", "a", "f", "x", "z"]
print(list1.index("z"))
# En caso de que el elemento a buscar no exista se lanzará una excepción:
# ValueError: 'w' is not in list
print(list1.index("w"))
```

1


```
-----
ValueError                                Traceback (most recent call last)
<ipython-input-21-1e9f9174c01d> in <module>()
      5 # En caso de que el elemento a buscar no exista se lanzará una excepción:
      6 # ValueError: 'w' is not in list
----> 7 print(list1.index("w"))

ValueError: 'w' is not in list
```

Listas: resumen

Una **lista** es una colección de datos **mutable**, es decir, que permite añadir, modificar y eliminar elementos. Los elementos están ordenados. También permite elementos duplicados.

Es la estructura de datos que más se utiliza.

- 
1. Crear una lista
 2. Acceso a los elementos de una lista
 - 2.1 Acceder con el índice
 - 2.2 Acceder con índices negativos
 - 2.3 Acceder con rangos de índices
 3. Modificar valores
 4. Verificar si existe un elemento usando el operador `in`
 5. Obtener el número de elementos usando la función `len()`
 6. Añadir nuevos elementos `append()`
 7. Eliminar elementos
 - 7.1 Método `remove()`
 - 7.2 Método `pop()`
 - 7.3 `del`
 8. Borrar una lista
 9. Vaciar una lista
 10. Copiar una lista
 11. Unir dos listas
 - 11.1 operador `+`
 - 11.2 `append()`
 - 11.3 `extend()`
 12. El constructor `list()`
 13. Invertir el orden de los elementos de una lista con `reverse()`
 14. Contar el número de veces que aparece un elemento en una lista con `count()`
 15. Ordenar una lista con el método `sort()`
 16. Averiguar el índice de un elemento

ESTRUCTURAS DE DATOS: TUPLAS



Tuplas

Una tupla es una colección de datos **ordenada** pero **inmutable**, es decir, no permite añadir, cambiar, borrar o reasignar elementos.

De esta forma, las **tuplas** son estáticas mientras que las **listas** son dinámicas.

Normalmente se utilizan **listas** para datos homogéneos y **tuplas** para datos heterogéneos.

- 1. Crear una tupla
- 2. Acceso a los elementos de una tupla
 - 2.1 Acceder con el índice
 - 2.2 Acceder con índices negativos
 - 2.3 Acceder con rangos de índices
- 3. Modificar valores (requiere convertir a lista, no se puede directamente)
- 4. Verificar si existe un elemento usando el operador `in`
- 5. Obtener el número de elementos usando la función `len()`
- 6. Crear una tupla de un solo elemento
- 7. Eliminar la tupla
- 8. Unir dos tuplas
- 9. El constructor tupla
- 10. El método `count(x)` devuelve el número de veces que `x` aparece en una tupla
- 11. Averiguar el índice de un elemento

Tuplas

Las tuplas se crean utilizando **paréntesis**.

Para acceder a los elementos de una tupla utilizamos **corchetes** y el número correspondiente al índice del elemento, es decir, a su posición, empezando siempre en 0.

```
# 1. Crear una tupla, utilizamos paréntesis  
# y dentro introducimos los elementos  
cars = ("tesla", "rolls", "ferrari")  
print(cars)
```

```
['tesla', 'rolls', 'ferrari']
```

```
# 2. Acceder a elementos de una tupla  
# 2.1 Para acceder a un elemento de la tupla lo hacemos  
# a través de su índice entre los corchetes,  
# empezando siempre en 0  
cars = ("tesla", "rolls", "ferrari")  
print(cars[0])  
print(cars[1])  
print(cars[2])
```

```
tesla  
rolls  
ferrari
```

Tuplas

También podemos acceder a sus elementos haciendo uso de **índices negativos** o **rangos de índices**.

Los **rangos de índices** nos devuelven una nueva tupla con los elementos según las posiciones especificadas.

```
# 2.2 También podemos acceder a los elementos haciendo uso de índices negativos
# De forma que -1 accede al 1 elemento empezando por el final
# De esta forma -2 accede al 2 elemento elemento por el final y así
cars = ("tesla", "rolls", "ferrari")
print(cars[-1])
print(cars[-2])
print(cars[-3])
```

```
ferrari
rolls
tesla
```

```
# 2.3 Otra forma de acceder es mediante rangos de índices
#           0         1         2         3         4         5         6
# Negativo: -7        -6        -5        -4        -3        -2        -1
cars = ("tesla", "rolls", "ferrari", "jaguar", "maserati", "audi", "porsche")
print(cars[2:5])
# Desde el primer índice indicado (incluido) hasta el último indicado (no incluido)
print(cars[2:5])
# Si no se especifica índice inicial por defecto se utilizará el primero de todos, el 0
print(cars[:4])
# Si no se especifica índice final por defecto se utilizará el último incluido
print(cars[2:])
# También aplican los índices negativos en los rangos.
# Acordarse de que el segundo índice, el final, no está incluido
print(cars[-4:-1])
```

```
('ferrari', 'jaguar', 'maserati')
('ferrari', 'jaguar', 'maserati')
('tesla', 'rolls', 'ferrari', 'jaguar')
('ferrari', 'jaguar', 'maserati', 'audi', 'porsche')
('jaguar', 'maserati', 'audi')
```

Tuplas

Las tuplas son **inmutables**, es decir, no se pueden modificar.

Una forma de 'modificar' una tupla es crear una lista a partir de la tupla, modificar la lista y después crear una nueva tupla a partir de la lista

```
# 3. Modificar valores
# Las tuplas no pueden modificarse, una manera de hacerlo es crear una lista
# a partir de una tupla, modificar la lista,
# y crear una nueva tupla a partir de la lista
x = ("tesla", "rolls", "ferrari")
y = list(x)
y[1] = "mercedes"
x = tuple(y)
print(x)
```

```
('tesla', 'mercedes', 'ferrari')
```

```
# 4. Comprobar si un elemento existe en una tupla.
# Se usa el operador in
cars = ("tesla", "rolls", "ferrari")
check = "apple" in cars
print(check)
```

```
False
```

Tuplas

Al igual que en el resto de colecciones de datos, para obtener la longitud o el número de elementos en la colección utilizamos la función **len()**.

Si necesitamos crear una tupla de un elemento será necesario especificar una **coma**.

```
# 5. Obtener la longitud de la tupla.  
# Se usa el método len()  
cars = ("tesla", "rolls", "ferrari")  
print(len(cars))
```

3

```
# 6. Crear una tupla de un sólo elemento  
cars = ("apple",)  
print(type(cars))  
  
# Sin la coma no es una tupla  
cars = ("apple")  
print(type(cars))
```

```
<class 'tuple'>  
<class 'str'>
```

Tuplas



No podemos **borrar** elementos de una tupla, pero podemos borrar la tupla con la palabra reservada **del**.

Para **unir** dos tuplas utilizamos el operador aritmético suma **+**.

```
# 7. Eliminar la tupla
cars = ("tesla", "rolls", "ferrari")
del cars
print(cars)
```

NameError Traceback
<ipython-input-13-6577de5bbdf9> in <module>()
 2 cars = ("tesla", "rolls", "ferrari")
 3 del cars
----> 4 print(cars)

NameError: name 'cars' is not defined

```
# 8. Unir dos tuplas
tuple1 = ("a", "b", "c")
tuple2 = (1, 2, 3)
tuple3 = tuple1 + tuple2
print(tuple3)
```

('a', 'b', 'c', 1, 2, 3)

Tuplas

Otra forma de crear una tupla es siguiendo los principios de la programación orientada a objetos mediante **constructores**.

Para obtener el número de veces que un elemento se repite en una tupla utilizamos la función **count**.

```
# 9. El constructor tupla
cars = tuple(("tesla", "rolls", "ferrari"))
print(cars)
```

```
('tesla', 'rolls', 'ferrari')
```

```
# 10. El método count nos devuelve el número de veces
# que un valor aparece en una lista
cars = ("a", "b", "c")
print(cars.count("a"))
cars = ("a", "b", "c", "a")
print(cars.count("a"))
print(cars)
```

```
1
2
('a', 'b', 'c', 'a')
```


Tuplas

Para obtener el índice o la posición en la que se encuentra un elemento utilizamos la función **index**.

En caso de que no exista el elemento en la tupla python lanza una **excepción**.

```
# 11. También podemos averiguar el índice de un elemento a partir de su valor
# IMPORTANTE: solo devuelve el índice de la primera ocurrencia encontrada
cars = ("b", "z", "d", "a", "f", "x", "z")
print(cars.index("z"))
# En caso de que el elemento a buscar no exista se lanzará una excepción:
# ValueError: tuple.index(x): x not in tuple
print(cars.index("w"))
```

1

```
-----
ValueError                                Traceback (most recent call last)
<ipython-input-19-2d45677fbd62> in <module>()
      5 # En caso de que el elemento a buscar no exista se lanzará una excepción:
      6 # ValueError: tuple.index(x): x not in tuple
----> 7 print(cars.index("w"))


ValueError: tuple.index(x): x not in tuple
```

Tuplas: resumen

Una tupla es una colección de datos **ordenada** pero **inmutable**, es decir, no permite añadir, cambiar, borrar o reasignar elementos.

De esta forma, las **tuplas** son estáticas mientras que las **listas** son dinámicas.

Normalmente se utilizan **listas** para datos homogéneos y **tuplas** para datos heterogéneos.

- 
1. Crear una tupla
 2. Acceso a los elementos de una tupla
 - 2.1 Acceder con el índice
 - 2.2 Acceder con índices negativos
 - 2.3 Acceder con rangos de índices
 3. Modificar valores (requiere convertir a lista, no se puede directamente)
 4. Verificar si existe un elemento usando el operador `in`
 5. Obtener el número de elementos usando la función `len()`
 6. Crear una tupla de un solo elemento
 7. Eliminar la tupla
 8. Unir dos tuplas
 9. El constructor tupla
 10. El método `count(x)` devuelve el número de veces que `x` aparece en una tupla
 11. Averiguar el índice de un elemento

ESTRUCTURAS DE DATOS: DICIONARIOS




Diccionarios

Un **diccionario** es un mapa de pares **clave valor**. Cada clave es un identificador único e inmutable, los valores son mutables.

Un ejemplo puede ser un diccionario cuyas claves sean el DNI del usuario y como valor los nombres y apellidos.

Lo más común a utilizar como claves con los identificadores de los registros de base de datos.

- 
1. Crear un diccionario
 2. Acceso a los elementos de un diccionario
 - 2.1 Acceder utilizando la clave
 - 2.2 Acceder utilizando el método `get()`
 3. Modificar valores
 4. Verificar si existe una clave usando el operador `in`
 5. Obtener el número de elementos usando la función `len()`
 6. Añadir nuevos elementos
 7. Eliminar elementos
 8. Borrar un diccionario
 9. Vaciar un diccionario
 10. Copiar un diccionario
 11. Diccionarios anidados
 12. El constructor `dict()`
 13. El método `items()` devuelve una lista con tuplas por cada clave-valor
 14. El método `keys()` para obtener una lista de claves
 15. El método `values()` para obtener una lista de valores

Diccionarios

Para crear un diccionario utilizamos llaves {}.

Dentro establecemos tantos pares clave-valor como queramos, separados por comas.

La **clave** se separa por : de su **valor**.

Para acceder a un elemento utilizamos corchetes y su clave.

```
# 1. Para crear un diccionario utilizamos llaves {}  
# haciendo uso de pares clave:valor  
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
print(thisdict)
```

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964}
```

```
# 2. Acceder a elementos de un diccionario  
# 2.1 Para acceder a un elemento utilizamos  
# la clave de ese elemento  
cars = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
x = cars["model"]  
print(x)
```

```
Mustang
```

Diccionarios

Otra forma de acceder a un par clave valor es mediante el método **get()** y la **clave**.

Para **modificar** valores lo hacemos igual que en una lista, con el operador asignación **=** asignamos un valor a una clave

```
# 2.2 Otra opción para acceder a un elemento
# es utilizar el método get()
x = cars.get("model")
print(x)
# Si la clave no existe lanzará un error, para evitarlo
# establecemos un valor por defecto como segundo parámetro
x = cars.get("cc", 0)
print(x)
```

```
Mustang
0
```

```
# 3. Modificar valores
cars = {
    "brand": "Ford",
    "model": "Mustang",
    "year": 1964
}
print(cars)
cars["year"] = 2018
print(cars)
```

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964}
{'brand': 'Ford', 'model': 'Mustang', 'year': 2018}
```

Diccionarios

Para verificar si existe una clave en un diccionario utilizamos el operador de membresía **in**.

Para obtener la longitud o número de pares clave valor en el diccionario utilizamos el método **len()**.

```
# 4. Verificar si existe una clave utilizamos el operador in
cars = {
    "brand": "Ford",
    "model": "Mustang",
    "year": 1964
}
if "model" in cars:
    print("Yes, 'model' is one of the keys in the cars dictionary")
```

Yes, 'model' is one of the keys in the cars dictionary

```
# 5. Obtener el número de elementos (pares clave valor)
cars = {
    "brand": "Ford",
    "model": "Mustang",
    "year": 1964,
    "cc": 5.1
}
print(len(cars))
```

4

Diccionarios

Los **diccionarios** son **mutables**, por tanto podemos añadir y eliminar elementos.

Para añadir un elemento usamos el operador **asignación** de forma que asignamos un **valor** a una **clave**.

Para eliminar un elemento de un diccionario hay múltiples maneras:

- pop()
- popitem()
- del

```
# 6. Añadir nuevos elementos a un diccionario
cars = {
    "brand": "Ford",
    "model": "Mustang",
    "year": 1964
}
cars["color"] = "red"
cars["cc"] = 5.1
print(cars)
```

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964, 'color': 'red', 'cc': 5.1}
```

```
# 7. Eliminar elementos de un diccionario
# 7.1 - Uso del método pop()
cars = {
    "brand": "Ford",
    "model": "Mustang",
    "year": 1964
}
cars.pop("model")
print(cars)
```

```
{'brand': 'Ford', 'year': 1964}
```


Diccionarios

Para eliminar un elemento de un diccionario hay múltiples maneras:

- pop()
- popitem()
- del

```
# 7.2 - Uso del método popitem()
cars = {
    "brand": "Ford",
    "model": "Mustang",
    "year": 1964
}
cars.popitem()
print(cars)
```

```
{'brand': 'Ford', 'model': 'Mustang'}
```

```
# 7.3 - Uso de la palabra reservada del
cars = {
    "brand": "Ford",
    "model": "Mustang",
    "year": 1964
}
del cars["model"]
print(cars)
```

```
{'brand': 'Ford', 'year': 1964}
```

Diccionarios

Para **borrar** un diccionario completamente utilizamos la palabra reservada **del**. Una vez eliminado ya no existe en memoria, si queremos volver a utilizarlo será necesario crear uno nuevo.

Para **eliminar** todos los elementos de un diccionario sin borrar el diccionario lo que hacemos es vaciarlo, para ello usamos el método `clear`, el diccionario sigue existiendo.

```
# 8. Borrar un diccionario por completo
cars = {
    "brand": "Ford",
    "model": "Mustang",
    "year": 1964
}
del cars
print(cars)
```

NameError
<ipython-input-24-f7ad2f70dd23> in <modul
6 }
7 del cars
----> 8 print(cars)

NameError: name 'cars' is not defined

```
# 9. Vaciar un diccionario
cars = {
    "brand": "Ford",
    "model": "Mustang",
    "year": 1964
}
print(cars)
cars.clear()
print(cars)
```

{'brand': 'Ford', 'model': 'Mustang', 'year': 1964}
{}

Diccionarios

Si queremos **duplicar** un diccionario usaremos el método **copy()**, esto genera una réplica con los mismos pares clave valor, pero diferentes en memoria por lo que la modificación de un diccionario no afecta al otro.

También podemos utilizar el **constructor dict()**.

```
# 10. Copiar un diccionario
# 10.1 El método copy()
# IMPORTANTE: dict2 = dict1 lo que hace es crear una nueva referencia a dict1
# por lo que los cambios aplicados a dict2 afecta a dict1, es por eso que para
# copiar utilizamos el método copy()
cars = {
    "brand": "Ford",
    "model": "Mustang",
    "year": 1964
}
mydict = cars.copy()
print(mydict)
```

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964}
```

```
# 10. Copiar un diccionario
# 10.2 El constructor dict para copiar
cars = {
    "brand": "Ford",
    "model": "Mustang",
    "year": 1964
}
mydict = dict(cars)
print(mydict)
```

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964}
```

Diccionarios

Es común que un diccionario pueda utilizar como valor otro diccionario. A esto se le llama **diccionarios anidados**.

```
# 11. Diccionarios anidados

# 11.1 Diccionario cuyos valores son otros diccionarios
myfamily = {
    "child1" : {
        "name" : "Emil",
        "year" : 2004
    },
    "child2" : {
        "name" : "Tobias",
        "year" : 2007
    },
    "child3" : {
        "name" : "Linus",
        "year" : 2011
    }
}
print(myfamily)
```

```
{'child1': {'name': 'Emil', 'year': 2004}, 'child2': {'name': 'Tobias', 'year': 2007}, 'child3': {'name': 'Linus', 'year': 2011}}
```

Diccionarios

Otra forma un poco más intuitiva de trabajar con diccionarios anidados es crear los diccionarios **de forma independiente** y después utilizar sus identificadores al crear el diccionario que los almacene.

```
# 11. Diccionarios anidados
# 11.2 Diccionario cuyos valores son otros diccionarios, expresado de forma
# más intuitiva, separando los diccionarios

child1 = {
    "name" : "Emil",
    "year" : 2004
}
child2 = {
    "name" : "Tobias",
    "year" : 2007
}
child3 = {
    "name" : "Linus",
    "year" : 2011
}

myfamily = {
    "child1" : child1,
    "child2" : child2,
    "child3" : child3
}

print(myfamily)
```

❏ {'child1': {'name': 'Emil', 'year': 2004}, 'child2': {'name': 'Tobias', 'year': 2007}, 'child3': {'name': 'Linus', 'year': 2011}}

Diccionarios

Otra forma de crear una tupla es siguiendo los principios de la programación orientada a objetos mediante **constructores**.

El método **fromkeys()** nos permite crear un nuevo diccionario a partir de las claves de un diccionario ya existente.

```
# 12. El constructor dict()
cars = dict(brand="Ford", model="Mustang", year=1964)
print(cars)
```

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964}
```

```
# 13. El método dict.fromkeys(keys, value) nos permite crear un diccionario
# con diferentes claves y el mismo valor para cada una
x = ('key1', 'key2', 'key3')
y = 0
cars = dict.fromkeys(x, y)
print(cars)

x = ('key1', 'key2', 'key3')
cars = dict.fromkeys(x)
print(cars)
```

```
{'key1': 0, 'key2': 0, 'key3': 0}
{'key1': None, 'key2': None, 'key3': None}
```

Diccionarios

El método **items()** nos devuelve un objeto vista el cual es una lista compuesta por tuplas. Cada tupla es un par clave valor del diccionario.

Los cambios realizados en el diccionario afectan al objeto vista.

```
# 14. El método items()
car = {
    "brand": "Ford",
    "model": "Mustang",
    "year": 1964
}
x = car.items()
print(x)

car = {
    "brand": "Ford",
    "model": "Mustang",
    "year": 1964
}
x = car.items()
car["year"] = 2018
print(x)
```

dict_items([('brand', 'Ford'), ('model', 'Mustang'), ('year', 1964)])
dict_items([('brand', 'Ford'), ('model', 'Mustang'), ('year', 2018)])

Diccionarios

El método **keys()** nos devuelve un objeto vista, que es una lista cuyos elementos son las **claves** del diccionario.

El método **values()** nos devuelve un objeto vista, que es una lista cuyos elementos son los **valores** del diccionario.

```
# 15. El método keys() nos devuelve una lista con las claves
car = {
    "brand": "Ford",
    "model": "Mustang",
    "year": 1964
}
x = car.keys()
print(x)
```

```
dict_keys(['brand', 'model', 'year'])
```

```
# 16. El método values() nos devuelve una lista con los valores
car = {
    "brand": "Ford",
    "model": "Mustang",
    "year": 1964
}
x = car.values()
print(x)
# Los cambios en el diccionario se transmiten a la lista
cars["year"] = 2020
print(x)
```

```
dict_values(['Ford', 'Mustang', 1964])
dict_values(['Ford', 'Mustang', 1964])
```


ESTRUCTURAS DE DATOS: CONJUNTOS



Conjuntos (Sets)

Un **set** o **conjunto** es una colección de elementos desordenada, sin índices, no admiten duplicados.

Se utilizan cuando el orden los datos no es importante, no queremos duplicados y necesitamos realizar operaciones como unión, intersección, etc.

- 1. Crear un `set`
- 2. Acceso a los elementos de un `set`
 - 2.1 Acceder con mediante estructuras de control: bucles
- 3. Modificar valores (no se puede)
- 4. Verificar si existe un elemento usando el operador `in`
- 5. Obtener el número de elementos usando la función `len()`
- 6. Añadir nuevos elementos
 - 6.1 Añadir un elemento
 - 6.2 Añadir más de un elemento a la vez
- 7. Eliminar elementos de un conjunto
 - 7.1 Eliminar con el método `remove()`
 - 7.2 Eliminar con el método `discard()`
 - 7.3 Eliminar con el método `pop()`
- 8. Eliminar un conjunto haciendo uso de `del`
- 9. Vaciar un conjunto con el método `clear()`
- 10. Copiar un conjunto con el método `copy()`
- 11. Unir dos conjuntos
 - 11.1 Operador `+`
 - 11.2 Método `union()`
 - 11.3 Método `update()`
- 12. El constructor `set()`
- 13. El método `difference()`
- 14. El método `difference_update()`
- 15. El método `intersection()`
- 16. El método `intersection_update()`
- 17. El método `isdisjoint()`
- 18. El método `issubset()`
- 19. El método `issuperset()`
- 20. El método `symmetric_difference()`
- 21. El método `symmetric_difference_update()`

Conjuntos (Sets)

Para crear un conjunto utilizamos llaves {}

Para acceder a los elementos de un conjunto utilizaremos estructuras iterativas, como el **bucle for**.

```
# 1. Para crear un conjunto utilizamos llaves {}  
# y dentro introducimos los elementos  
cars = {"tesla", "rolls", "ferrari"}  
print(cars)
```

```
{'rolls', 'tesla', 'ferrari'}
```

```
# 2. Para acceder a elementos de un set no podemos usar  
# el índice de cada elemento, necesitaremos utilizar  
# estructuras de repetición: for, while  
cars = {"tesla", "rolls", "ferrari"}  
for car in cars:  
    print(car)  
  
# print(cars[0]) # Da error TypeError: 'set' object is not subscriptable
```

```
rolls  
tesla  
ferrari
```

Conjuntos (Sets)

Los elementos de un conjunto no se pueden **modificar**. Una forma de hacerlo es eliminar un elemento y añadir uno nuevo.

Para comprobar si un elemento **existe** en un conjunto utilizamos el operador de membresía **in**.

```
# 3. Una vez creado el conjunto no se pueden editar sus items
cars = {"tesla", "rolls", "ferrari"}
cars[1] = "blackcurrant"
print(cars)
```

```
-----
TypeError                                 Traceback (most recent call)
<ipython-input-6-60f1b995f755> in <module>()
      1 # 3. Una vez creado el conjunto no se pueden editar sus items
      2 cars = {"tesla", "rolls", "ferrari"}
----> 3 cars[1] = "blackcurrant"
      4 print(cars)

TypeError: 'set' object does not support item assignment
```

```
# 4. Comprobar si un elemento existe en una conjunto.
# Se usa el operador in
cars = {"tesla", "rolls", "ferrari"}
check = "rolls" in cars
print(check)
print("mercedes" in cars)
```

```
True
False
```

Conjuntos (Sets)

Para obtener la longitud o número de elementos en un conjunto utilizamos la función **len()**.

```
# 5. Obtener longitud de conjunto.  
# Se usa el método len()  
cars = {"tesla", "rolls", "ferrari"}  
print(len(cars))
```

3

Para añadir un nuevo elemento a un conjunto existen varias formas:

- Método **add()**
- Método **update()**

```
# 6. Añadir nuevos elementos al conjunto  
# 6.1 Añadir un elemento al conjunto. Se usa el método add()  
cars = {"tesla", "rolls", "ferrari"}  
cars.add("mercedes")  
print(cars)
```

{'rolls', 'tesla', 'ferrari', 'mercedes'}

Conjuntos (Sets)

Con el método **update()** podemos agregar los elementos de otra estructura de datos al conjunto.

```
# 6.2 Para añadir mas de 1 elemento a conjunto.  
# Se usa el método update()  
# podemos añadir tanto sets, como listas como tuplas  
cars = {"tesla", "rolls", "ferrari"}  
cars.update(["jaguar", "maserati", "audi", "porsche"]) # se añade una lista  
print(cars)  
cars = {"tesla", "rolls", "ferrari"}  
cars.update({"jaguar", "maserati", "audi", "porsche"}) # se añade un set  
print(cars)
```

```
{'maserati', 'jaguar', 'porsche', 'audi', 'rolls', 'tesla', 'ferrari'}  
{'audi', 'rolls', 'tesla', 'maserati', 'ferrari', 'jaguar', 'porsche'}
```

Conjuntos (Sets)

Para borrar elementos de un conjunto utilizamos:

- **remove()**: si el elemento no existe entonces lanzará una excepción.
- **discard()**: borra el elemento únicamente si existe. No lanza error en caso de que no exista.
- **pop()**: borra un elemento cualquiera.

```
# 7. Borrar elementos del conjunto
# 7.1 Métodos remove()
cars = {"tesla", "rolls", "ferrari"}
print(cars)
cars.remove("tesla")
print(cars)
```

```
{'rolls', 'tesla', 'ferrari'}
{'rolls', 'ferrari'}
```

```
# 7.2 Método discard()
cars = {"tesla", "rolls", "ferrari"}
print(cars)
cars.discard("tesla")
print(cars)
```

```
{'rolls', 'tesla', 'ferrari'}
{'rolls', 'ferrari'}
```

```
# 7.3 El método pop()
# pop: Eliminar un elemento, devuelve el que se haya borrado,
# no tiene por qué ser el último ya que el set no tiene orden
cars = {"tesla", "rolls", "ferrari"}
x = cars.pop()
print(x)
print(cars)
```

```
rolls
{'tesla', 'ferrari'}
```

Conjuntos (Sets)

Con la palabra reservada **del** podemos borrar el set.

Con el método **clear()** podemos vaciar el set.

```
# 8. Eliminar un conjunto
# del: eliminarconjunto
cars = {"tesla", "rolls", "ferrari"}
del cars
print(cars)
```

```
-----
NameError                                Traceback
<ipython-input-26-7c95b7476463> in <module>()
      3 cars = {"tesla", "rolls", "ferrari"}
      4 del cars
----> 5 print(cars)

NameError: name 'cars' is not defined
```

```
# 9. Vaciar una conjunto
cars = {"tesla", "rolls", "ferrari"}
cars.clear()
print(cars)
```

```
set()
```


Conjuntos (Sets)

Para copiar un set utilizamos el método **copy()**.

Para unir dos conjuntos podemos utilizar el método **union()** o **update()**.

La diferencia está en qué union devuelve un nuevo set, mientras que update actualiza el existente.

```
# 10. Copiar un conjunto
cars = {"tesla", "rolls", "ferrari"}
carsCopy = cars.copy()
print(carsCopy)
```

```
{'rolls', 'tesla', 'ferrari'}
```

```
# 11. Unir dos conjuntos diferentes
# 11.1 union()
set1 = {"a", "b", "c"}
set2 = {1, 2, 3}
set3 = set1.union(set2)
print(set3)
```

```
{1, 'a', 2, 3, 'b', 'c'}
```

```
# 11. Unir dos conjuntos diferentes
# 11.2 update()
set1 = {"a", "b", "c"}
set2 = {1, 2, 3}
set1.update(set2)
print(set1)
```

```
{1, 'a', 2, 3, 'b', 'c'}
```

Conjuntos (Sets)

Otra forma de crear un conjunto es siguiendo los principios de la programación orientada a objetos mediante **constructores**.

El método **difference()** nos permite obtener un nuevo conjunto con aquellos elementos no comunes a ambos set y que existan solo en el primero pero no en ambos.

```
# 12. Constructor conjunto
cars = set(("tesla", "rolls", "ferrari"))
print(cars)
```

```
{'rolls', 'tesla', 'ferrari'}
```

```
# 13. El método difference() nos devuelve un nuevo conjunto
# con los elementos que no estén contenidos en ninguno de dos sets
x = {"tesla", "rolls", "ferrari"}
y = {"mercedes", "rolls", "jaguar"}
z = x.difference(y)
print(z)
```

```
{'tesla', 'ferrari'}
```

Conjuntos (Sets)

El método **difference_update()** actualiza el set actual eliminando aquellos elementos que haya en uno o más sets.

```
# 14. difference_update() Elimina de un set que ya existe  
# los elementos que existan en dos conjuntos  
x = {"tesla", "rolls", "ferrari"}  
y = {"mercedes", "rolls", "jaguar"}  
x.difference_update(y)  
print(x)
```

```
{'tesla', 'ferrari'}
```

El método **intersection()** crea un nuevo set compuesto por aquellos elementos comunes a los sets sobre los que se aplica.

```
# 15. intersection() nos devuelve un nuevo set con aquellos elementos  
# que existan en común en dos conjuntos  
x = {"tesla", "rolls", "ferrari"}  
y = {"mercedes", "rolls", "ferrari"}  
z = x.intersection(y)  
print(z)
```

```
{'rolls', 'ferrari'}
```

Conjuntos (Sets)

El método **intersection_update()** actualiza el primer set con aquellos elementos comunes a todos los sets sobre los que aplica.

El método **isdisjoint()** devolverá True cuando no haya elementos comunes entre los set y devolverá False cuando haya al menos un elemento en común.

```
# 16. intersection_update() elimina los items del primer set  
# que no sean comunes a ambos sets  
x = {"tesla", "rolls", "ferrari"}  
y = {"mercedes", "rolls", "jaguar"}  
x.intersection_update(y)  
print(x)
```

```
{'rolls'}
```

```
# 17. isdisjoint() devuelve True o False si ningún elemento del  
# primer conjunto aparece en el segundo conjunto  
x = {"tesla", "rolls", "ferrari"}  
y = {"mercedes", "rolls", "jaguar"}  
z = x.isdisjoint(y)  
print(z)  
x = {"tesla", "rolls", "ferrari"}  
y = {"mercedes", "subaru", "jaguar"}  
z = x.isdisjoint(y)  
print(z)
```

```
False  
True
```

Conjuntos (Sets)

El método **issubset()** nos devuelve **True** si el set actual es parte de otro set, es decir, los elementos del primero están en el segundo, en caso contrario devuelve **False**.

En método **superset()** actúa al revés, nos devuelve **True** si los elementos del segundo set están contenidos en el primero, en caso contrario devuelve **False**.

```
# 18. issubset() Devuelve True si todos los elementos del  
# primer conjunto aparecen en el segundo conjunto  
x = {"a", "b", "c"}  
y = {"f", "e", "d", "c", "b", "a"}  
z = x.issubset(y)  
print(z)
```

True

```
# 19. issuperset() Devuelve True si todos los elementos del  
# segundo conjunto aparecen en el primer conjunto  
x = {"f", "e", "d", "c", "b", "a"}  
y = {"a", "b", "c"}  
z = x.issuperset(y)  
print(z)
```

True

Conjuntos (Sets)

El método

`symmetric_difference()` devuelve un nuevo conjunto con aquellos elementos que no sean comunes a los sets que aplica.

En método

`symmetric_difference_update()` realiza lo mismo pero en vez de devolver un nuevo conjunto lo que hace es actualizar el conjunto sobre el que aplica.

```
# 20. symmetric_difference() Devuelve un nuevo conjunto con  
# aquellos elementos que no son comunes a ambos conjuntos  
x = {"tesla", "rolls", "ferrari"}  
y = {"google", "microsoft", "apple"}  
z = x.symmetric_difference(y)  
print(z)
```

```
{'google', 'apple', 'ferrari', 'rolls', 'tesla', 'microsoft'}
```

```
# 21. symmetric_difference_update() Elimina aquellos elementos que están presentes  
# en ambos conjuntos, y añade al primer set aquellos que no estén presentes en ambos  
x = {"tesla", "rolls", "ferrari"}  
y = {"mercedes", "subaru", "jaguar"}  
x.symmetric_difference_update(y)  
print(x)
```

```
{'mercedes', 'jaguar', 'subaru', 'rolls', 'tesla', 'ferrari'}
```

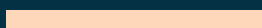
Sets: resumen

Un **set** o **conjunto** es una colección de elementos desordenada, sin índices, no admiten duplicados.

Se utilizan cuando el orden los datos no es importante, no queremos duplicados y necesitamos realizar operaciones como unión, intersección, etc.

- 1. Crear un `set`
- 2. Acceso a los elementos de un `set`
 - 2.1 Acceder con mediante estructuras de control: bucles
- 3. Modificar valores (no se puede)
- 4. Verificar si existe un elemento usando el operador `in`
- 5. Obtener el número de elementos usando la función `len()`
- 6. Añadir nuevos elementos
 - 6.1 Añadir un elemento
 - 6.2 Añadir más de un elemento a la vez
- 7. Eliminar elementos de un conjunto
 - 7.1 Eliminar con el método `remove()`
 - 7.2 Eliminar con el método `discard()`
 - 7.3 Eliminar con el método `pop()`
- 8. Eliminar un conjunto haciendo uso de `del`
- 9. Vaciar un conjunto con el método `clear()`
- 10. Copiar un conjunto con el método `copy()`
- 11. Unir dos conjuntos
 - 11.1 Operador `+`
 - 11.2 Método `union()`
 - 11.3 Método `update()`
- 12. El constructor `set()`
- 13. El método `difference()`
- 14. El método `difference_update()`
- 15. El método `intersection()`
- 16. El método `intersection_update()`
- 17. El método `isdisjoint()`
- 18. El método `issubset()`
- 19. El método `issuperset()`
- 20. El método `symmetric_difference()`
- 21. El método `symmetric_difference_update()`

RESUMEN



Recordatorio

Usamos estructuras de datos cuando trabajamos con colecciones de elementos.

Hemos visto 4 estructuras de datos diferentes:

- Listas con **corchetes**:
 - `thislist = ["a", "b", "c"]`
- Tuplas con **paréntesis**:
 - `thistuple = ("a", "b", "c")`
- Diccionarios con **llaves y pares clave valor**:
 - `thisdictionary = {"clave1": "valor1", "clave2": "valor2", "clave3": "valor3"}`
- Conjuntos con **llaves**:
 - `thisset = {"a", "b", "c"}`